Data Palooza

- covering the internals of how many languages manage data (including types, variables and values)

Variables and Values

- **variable** - the symbolic name associated with a location that contains a value or a pointer
- **value** - a piece of data with a type (usually) that is either referred to by a variable, or computed by a program

What are the facets that make up a variable?

- *names*: variables almost always have a name
- *types*: a variable may (or may not) have a type associated with it
- *values*: a variable stores a value (and its type)
- *binding*: how a variable is connected to its current value
- *storage*: the slot of memory that stores the value associated with the variable
- *lifetime*: the timeframe over which a variable exists
- *scope*: when/where a variable is accessible by code
- *mutability*: can a variable's value be changed?

What are the facets that make up a value?

- ~~*names*: variables almost always have a name~~
- *types*: a value **will always** have a type associated with it
- ~~*values*: a variable stores a value (and its type)~~
- ~~*binding*: how a variable is connected to its current value~~
- *storage*: the slot of memory that stores the value
- *lifetime*: the timeframe over which a value exists
- ~~*scope*: when/where a variable is accessible by code~~
- *mutability*: can a value be changed?

> *Note: Lifetime and scope are **not** the same. Lifetime refers to the existence of the variable. Scope refers to the accessibility of a variable. It's possible for a variable to be out-of-scope but still be alive.*

Variable names

- Almost all languages stipulate that names should contain valid characters
- Almost all languages stipulate that names should not be the same as keywords or constants
- Most languages have a rule that disallows spaces in variable names
- Some languages have rules about special characters in names, some enforce length restrictions, and some even enforce some sort of case sensitivity rule.
- naming conventions important to enforce standardization

Variable storage

- Variables and values stored in one the stack, the heap, or the static data area.
- usually:
  - function parameters, local variables on stack
  - dynamically allocated variables on heap
  - static members, globals in static data area

- can have combinations (eg: pointers on stack pointing to heap)

Variable types

Variable lifetime and scope

Every variable and value has a lifetime, over which they are valid and may be accessed. Note that it is possible for a variable to be "alive" during execution, and yet not be accessible. Some languages give the programmer explicit control over the lifetime of a value, while others completely abstract this away.

```
void bar() {
  ...
}

int main() {
  int x = 5;
  bar();
  ...
}
```

A variable is in scope if it can be explicitly accessed by name in that region.

```
void bar(int *ptr) {
  cout << x;    // ERROR! x isn't in-scope here!
  cout << *ptr; // Even though its value can be accessed
}

void foo() {
  int x;
  cout << x;    // x is in-scope here
  bar(&x);
}
```

There are two primary approaches to scoping: lexical (or static) scoping and dynamic scoping. We will cover these topics in the next lecture.

Types

- What can you infer about a value, given its type?
  - The set of legal values it can hold
  - The operations we can perform on it
  - How much memory you need
  - How to interpret the bytes stored in RAM
  - How values are converted between types
- it's possible to have a language with no types, eg: assembly
- But virtually all modern languages have a type system, since it makes programming so much easier and safer.
- every variable doesn't need to a type in a type language
  - eg: in Python variables aren't associated with types

```
def foo(q):
  if q:
```

```
    x = "What's my type?" # string
  else:
    x = 10 # int
  ...
```

- many dynamically typed languages don't associate *variables* with types.
- a value is **always** associated with a type.
- **primitive types** - a set of types from which other types are constructed
  - Integers, booleans, characters, enumerated types (enums), floats, pointers
- **composite data types** - types which can be constructed using primitive and composite data types
  - strings, structs, variants/unions, objects, tuples, containers (arrays, sets, maps)
- data types that don't fall into either category
  - Functions, generic types, boxed types

Boxed Type

- **boxed type** - an object whose only data member is a *primitive*
- In languages like Python that pass by object reference, this lets you "change" a primitive type's value from within another function!

```
class Integer {
 public:
   int get() const { return val_; }
 private:
   int val_;
};
```

User Defined Types

- classes, structs, enums, interfaces, etc are user-defined
- when they're declared, language *implicitly* defines a new type

Value and Reference Types

- **value type** - can be used to instantiate objects/values *and* define pointers/object refs/references
- **reference types** - can only be used to define pointers/object refs/references, NOT instantiate objects/values
  - eg: abstract classes, interfaces

Dimensions of Typing

- Strictness: strong vs weak typing
- Compile-time vs Run-time: static vs dynamic

	Static	**Dynamic**
Strong	C#, Go, Haskell, Java	JS, Ruby, Python
Weak	Assembly, C, C++	...

Type checking

- **Type checking** - the process of verifying and enforcing constriants on types.
- can occur during compile time (static) or during run time (dynamic).
- the language can also specify the degree of strictness for type checking (strong vs weak type checking)

Static typing

- verifies all operations in a program are consistent with the types of the operands *prior* to execution
- types can be explicitly specified (C++) or can be inferred from code (Haskell)
- **type inference** - automatic detection of types of expressions or variables in a language
  - For inferred types, if the type checker cannot assign distinct types to all variables, functions, and expressions or cannot verify type safety, then it generates a compiler error
- variables have fixed type at assignment and maintain same type throughout lifetime
- if a program type checks, then the code is (largely) type safe, and few if any run time checks need to be performed
- note: to support static typing, a language **must have** a fixed type bound to each variable at the time of definition.

Type Inference

```
void foo(_____ x, _____ y) {
  cout << x + 10;
  cout << y + " is a string!";
}
```

compiler can infer `x` is `int` and `y` is `string`

```
void foo(_____ x, _____ y) {
  cout << x + 10;
  cout << y + " is a string!";
}
void bar() {
  double d = 3.14;
  foo(d,"barf");
}
```

Here, it makes more sense for `x` to be a `double`. So type inference is actually a complex constraint satisfaction problem.

Many statically typed languages now offer some form of type inference (`auto` in C++)

```
// C++ type inference with auto
int main() {
  auto x = 3.14159; // x is of type int
  ...

  vector<int> v;
  for (auto item: v) { // item is of type int
    cout << item << endl;
  }
```

```
  auto it = v.begin(); // it is of type vector<int>::iterator
  while(it != v.end()) {
    cout << *it << endl;
    ++it;
  }
}
```

- Note: Type inference has limitations! Most languages (like C++, Java, or Rust) will struggle with generic types and iterators; you'll be forced to write type annotations in some areas.

Go infers type with `:=` operator

```
// GoLang type inference
func main() {
  msg := "I like languages"; // string
  n := 5 // int
  for i := n; i > 0; i-- {
    fmt.Println(msg);
  }
}
```

Conservatism

Static type checking is inherently conservative--the static type checker will disallow valid programs that never violate typing rules.

Example

```
class Mammal {...};
class Dog: public Mammal {
public:
 void bite() { cout << "Chomp\n"; }
};
class Cat: public Mammal {
public:
 void scratch() { cout << "Scrape!\n"; }
};

void handlePet(Mammal& m, bool bite, bool scratch) {
 if (bite)
   m.bite();
 if (scratch)
   m.scratch();   // ERROR: no member named 'scratch' in 'Mammal'
}

int main() {
 Dog spot;
 Cat meowmer;
 handlePet(spot, true, false);
 handlePet(meowmer, false, true);
}
```

The compiler generates an error during the compilation of this code even though the code only asks `Dog`s to `bite` and `Cat`s to `scratch`.

Pros of static type checking:

- faster code (since we don't have to type check during run time)
- Allows for earlier bug detection (at compile time)
- No need for custom type checks

Cons of static type checking:

- Conservative--may error our perfectly valid code
- requires a type checking phase before execution, which can be slow

Dynamic typing

- verifying the type safety of a program at run time.
- if code is attempting an illegal operation on a variable's value, an exception is generated

```
def add(x,y):
 print(x + y)   # throws TypeError at runtime

def foo():
 a = 10
 b = "cooties"
 add(a,b)

# ==============================

def do_something(x):
 x.quack()     # throws AttributeError at runtime

def main():
 a = Lion("Leo")
 do_something(a)
```

- *variables* don't have types, *values* have types
- Most dynamically typed languages don't require explicit type annotations for variables
- a given variable name could refer to values of multiple types over the course of execution

Type Tags

- Usually the compiler/interpreter uses a **type tag** - an extra piece of data stored along with each value that indicates the type of the value.

```
def add(x,y):
 print(x + y)

def foo():
 a = 10
 b = "nerd"
 add(a, b)
```

- compiler would store that the value `a` is pointing to is an `int` with value `10`, and `b` is a `string` with value `nerd`.

Example:

```
v := 42.1
v = 42
fmt.Print(v)
fmt.Printf("; type: %T", v)
```

If statically typed, prints `42.0 type: float`

- `v` is declared as float and stays that way throughout its lifetime
- `42` gets implicitly cast (coerced) to float

If dynamically typed, prints `42 type: int`

- `v` is declared as float, but reassigned to int value, no casting/coverting necessary
- type associated with variable is the tag its value has

Note: this is Go which does neither of these things

Dynamic Type Checking in Statically-Typed Languages

- sometimes statically typed languages need to perform run time type checks.
  - Eg: when downcasting (in C++)
  - when disambiguating variants (think Haskell!)
  - (depending on the implementation) potentially in runtime generics
- **down-casting** - when casting an object from a superclass to a subclass

```
class Person { ... };
class Student : public Person { ... };
class Doctor : public Person { ... };
void partay(Person *p) {
  Student *s = dynamic_cast<Student *>(p);
  if (s != nullptr)
    s->getDrunkAtParty();
}
int main() {
  Doctor *d = new Doctor("Dr. Fauci");
  partay(d);
  delete d;
}
```

`dynamic_cast` does a run time check to ensure that the type conversion (from `Person*` to `Student*`) we are performing is valid. If it isn't valid, `dynamic_cast` will return `nullptr`.

Pros of dynamic type checking:

- Increased flexibility
- Often easier to implement generics that operate on many different types of data
- Simpler code due to fewer type annotations
- faster prototyping

Cons of dynamic type checking:

- detect errors much later
- slower code due to run time type checking
- Requires more testing for the same level of assurance
- No way to guarantee safety across all possible executions (unlike static type checking)

Duck Typing

- Duck typing emerges as a consequence of dynamic typing
- in static typing, determine the what operations work on a particular value/variable based on its type.
- but with dynamic typing, variables no longer have a fixed type, so we can pass a value of any type to a function, and as long as the type of the value implements all of the operations used in the function, the code should work
- so duck typing is the only option on dynamically-typed languages
- Note: also possible to have something similar to duck typing in statically typed languages
  - In C++, this is done using templates; there's also runtime generics in Java, Rust's generics system, etc

```
class Duck:
 def swim(self):
  print("Paddle paddle paddle")

class OlympicSwimmer:
 def swim(self):
  print("Back stroke back stroke")

class Professor:
 def profess(self):
  print("Blah blah blah blah blah")

def process(x):
 x.swim()

def main():
 d = Duck()
 s = OlympicSwimmer()
 p = Professor()
 process(d) # Paddle paddle paddle
 process(s) # Back stroke back stroke
 process(p) # throws AttributeError
```

Supporting enumeration

In Python, you can make any class iterable by implementing `__iter__` and `__next__` methods

```
# Python duck typing for iteration
class Cubes:
  def __init__(self, lower, upper):
    self.upper = upper
```

```
    self.n = lower
  def __iter__(self):
    return self
  def __next__(self):
    if self.n < self.upper:
      s = self.n ** 3
      self.n += 1
      return s
    else:
      raise StopIteration

for i in Cubes(1, 4):
  print(i)             # prints 1, 8, 27
```

Make any class printable!

In Python, you can make any class printable (using `print`) by implementing `__str__` function

```
# Python duck typing for printing objects
class Duck:
 def __init__(self, name, feathers):
  self.name = name
  self.feathers = feathers

 def __str__(self):
  return self.name + " with " + \
         str(self.feathers) + " feathers."

d = Duck("Daffy", 3)
print(d)
```

Make any class comparalbe!

In Python, if you add the `__eq__` method to any class, you can make it's objects "comparable"

```
# Python duck typing for equality
class Duck:
  def __init__(self, name, feathers):
    self.name = name
    self.feathers = feathers

 def __eq__(self, other):
  return (
    self.name == other.name and
    self.feathers == other.feathers
  )

duck1 = Duck("Carey", 19)
duck2 = Duck("Carey", 19)
```

```
if duck1 == duck2:
  print("Same!")
```

Gradual Typing

- hybrid approach between static and dynamic typing (used by PHP, Typescript)
- some variables may have explicitly annotated types, while others may not.
- allows type checker to partially verify type safety prior to execution, and perform the rest of the checks during run time.
    - If a variable is untyped, then type errors for that variable are detected at run time
    - if a variable is typed, then it's possible to detect some type errors at compile time

What if we pass an untyped variable as an argument to a typed variable?

```
def square(x: int):
  return x * x

def what_happens(y):
  print(square(y))   # y is untyped
```

- allowed in gradually typed languages!
- if you pass an untyped variable as an argument to `what_happens`, the type checker will check for errors during run time
- This way, if you do use an invalid type, the program will generate an error the moment an incompatible type is detected.

Strong typing

- strong typing ensures we will **never** have *undefined behaviour* at run time due to type issues
- all operations on variables will either succeed or generate an explicit type exception at runtime
- no possibility of an unchecked runtime type error
    - consistent exceptions are *not* undefined behavior

> ***NOTE:***
>
> - *definition of strong typing is disputed*
> - *Many academics argue for a stronger definition*
> - *e.g: all conversions between types should be explicit*
> - *the language should have explicit type annotations for all variables*
> - *while these definitions may make a languages type system stricter, they don't impact the languages type or memory safety.*

These are the minimum requirements for a language to be strongly typed:

- **the language is type-safe**: the language will prevent an operation on a variable `x` if `x`s type doesn't support that operation.

```
int x;
Dog d;
a = 5 * d; // Prevented
```

- **the language is memory safe**: the language prevents inappropriate memory accesses (e.g., out-of-bounds array accesses, accessing a dangling pointer)

```
int arr[5], *ptr;
cout << arr[10]; // Prevented
cout << *ptr;   d// Prevented
```

- can be enforced either statically or dynamically!

Languages usually use a few techniques to implement strong typing:

- before an expression is evaluated, the compiler/interpreter validates that all of the operands used in the expression have compatible types.
- all conversions between different types are checked and if the types are incompatible (e.g., converting an int to a Dog), then an exception will be generated.
- pointers are either set to null or assigned to point at a valid object at creation.
- accesses to arrays are bounds checked; pointer arithmetic is bounds-checked.
- the language ensures objects can't be used after they are destroyed.

In general, strongly typed languages prevent operations on incompatible types or invalid memory.

Why do we need memory safety?

Why do strongly typed langauges require memory safety? To answer this question, consider the following example in C++ (a weakly typed language).

```
// C++
int arr[3] = {10, 20, 30};
float salary = 120000.50;

cout << arr[3]; // out-of-bounds access
```

- `arr[3]` actually access the value stored in `salary`, since all local variables are stored on the stack.
- So if a language is not memory safe, then it's possible to access a value (like `salary`) using an invalid type (`int` instead of `float`).
- Accessing a dangling pointer is another example of how memory safety can violate type safety.

```
// Accessing a dangling pointer
float *ptr = new float[100];
delete [] ptr;
cout << *ptr; // is this still a float?
```

Checked type casts

- **checked cast** a type cast that throws an exception/error if the conversion is illegal

Java example:

```
public void petAnimal(Animal a) {
  Dog d = (Dog)a; // Runtime Exception: can't cast cat to dog
  d.wagTail();
}

...

public void takeCareOfCats() {
  Cat c = new Cat("Meowmer");
  petAnimal(c);
}
```

A similar snippet of code in C++ however would run, even though we are dealing with an object of type `Cat`, not `Dog`!

```
void petAnimal(Animal *a) {
  Dog* d = (Dog *)a;
  d->wagTail();
}

void takeCareOfCats() {
  Cat c("Meowmer");
  petAnimal(&c);
}
```

When the C++ program executes `d->wagTail()`, anything could happen--*undefined behaviour*

Advantages of Strong Typing

- dramatically reduce software vulnerabilities (e.g. buffer overflows).
- early detection and fixing of errors/bugs.

Why do people use weakly typed languages?

- performance (less checks)
- legacy

Weak typing

- **does not** guarantee that all operations are invoked on objects/values of the appropriate type
- Weakly typed languages are generally neither type safe nor memory safe.

Undefined behaviour

- in weakly-typed languages, we can have undefined behavior at runtime!
- **Undefined behaviour** - the result of executing a program whose behaviour is prescribed as unpredictable in the language spec.

Example: Calling `get_iq` on `n` (which actually an `integer`) is undefined behavior

```
// C++ example w/ undefined behavior!
class Nerd {
public:
  Nerd(string name, int IQ) { ...}
```

```
  int get_iq() { return iq_; }
  ...
};

int main() {
  int a = 10;
  Nerd *n = reinterpret_cast<Nerd *>(&a); // reinterpret int as Nerd object
  cout << n->get_iq(); // ?? What happens?!?!?
}
```

Tough to say whether or not a language is weakly or strongly typed just from looking at its behaviour in one situation

```
# Defines a function called ComputeSum
# @_ is an array that holds all arguments

sub ComputeSum {
   $sum = 0;

   foreach $item (@_) {    # loop thru args
      $sum += $item;
   }

   print("Sum of inputs: $sum\n")
}

# Function call
ComputeSum(10, "90", "cat");
```

This outputs

```
Sum of inputs: 100
```

- may seem weakly typed, but is strongly typed!
- in Perl, the `+` operator does support the `string` type
  - converts the `string` to an `integer`, if possible (e.g., if it holds all digits)
  - If not, convert the `string` to 0
- so while Perl does perform implicit conversions which might not do what you want, its behavior is **never undefined**.
- Industry has trended towards *strong, statically typed* languages

Subtypes and Supertypes

Given two types T_{sub} and T_{super}, T_{sub} is a subtype of T_{super} iff

1. every element belonging to the set of values of type T_{sub} is also a member of the set of values of T_{super}.
2. *All* operations (eg +, -, *, /) that you can use on a value of type T_{super} must also work properly on a value of type T_{sub}.

- i.e., If I have code designed to operate on a value of type T_{super}, it must also work if I pass in a value of type T_{sub}.

Examples:

- `short` is a subtype of `int`
- `float` is subtype of `double` and `double` is supertype of float
- `int` is a subtype of `const int` because all operations that can be performed on a `const int` can also be performed on an `int` and they have the same set of values.
  - `const int` is NOT a subtype of `int` because `const int` can't be reassigned like `int` can
- `int` is NOT a subtype of `float` but `int` IS (sorta) a subtype of `double` since `double`s have enough precision to represent all values that `int` can hold.

```
class Person {
public:
  virtual void eat();
};

class Nerd: public Person {
  virtual void eat();
  virtual void study();
};
```

- `Nerd` is a subtype of `Person`. All operations that can be performed on a `Person` can also be performed on a `Nerd` **and** all `Nerd`s belong to the set of `Person`s.

With Inheritance

- base class and subclass both define a new type (Person and Nerd)
- Nerd is a subclass of Person, so Nerd type is subclass of Person type

Type Casting, Conversion

- Type conversion and type casting are used when we want to perform an operation on a value of type A, but the operation requires a value of type B

Type conversions

- takes a value of type A and generates a whole new value (occupying new storage, with a different bit encoding) of type B.
- typically used to convert between primitives (e.g. float and int).

```
void convert() {
  float pi = 3.141;
  cout << (int)pi; // compiler generates new value of diff type
}
```

Type casts

- A cast takes a value of type A and treats it as if it were value of type B – no conversion takes place! No new value is created!
- typically used with objects.

```
// treat int as unsigned int!
int main() {
```

```
  int val = -42;

  cout << (unsigned int)val;
      // prints 4294967254
}
```

`val` refers to the original `int` object, but interprets the bits as if it were an `unsigned int`.

Explicit vs Implicit

- **explicit conversion/cast** requires the programmer to use explicit syntax to force the conversion/cast
- **implicit conversion/cast** happens without explicit conversion syntax
  - **coercion** - implicit conversion

Explicit Conversions/Casts

- explicity tell the compiler to change what would be a compile time error to a run time check.

```
class Person { ... }
class Student extends Person { ... }
class Professor extends Person { ... }

class Example
{
  public void do_your_thing(Professor q) {
    q.give_a_lecture();
  }
  public void process_person(Person p) {
    if (p.get_name() == "Carey")
      do_your_thing(p);   // implicit conversion; Person to Professor
  }
}
```

- programmer may know that if `p`'s name is Carey, p must be a Prof, so the code is safe, but statically-typed compiler can't
- compiler outputs an error, need explicit cast to tell compiler to allow

```
  public void process_person(Person p) {
    if (p.get_name() == "Carey")
      do_your_thing((Professor)p);   // explicit
  }
```

- if the language is strongly typed, then it will perform a runtime check to ensure the type conversion is valid
- in weakly typed languages, improper casts/conversions are often not checked at run time, leading to nasty bugs

Explicit conversions:

```
// Explicit C++ conversions
float fpi = 3.14;
```

```
int ipi = (int)fpi;               // old way
int ipi2 = static_cast<int>(fpi); // new way
```

Note: although `static_cast<int>(fpi)` is creating a new value (so it's performing a conversion), C++ calls it a cast

```
# Explicit Python conversions
fpi = 3.14
ipi = int(pi)
```

Explicit Casts:

```
// Explicit C++ cast
class Person { ... };
class Student: public Person { ... }

void make_em_study(Person *p) {
  Student *s = dynamic_cast<Student*>(p);
  if (s != nullptr)
    s->study();
}
```

Implicit conversions and casts

- Most languages have a set of rules that govern implicit conversions that may occur without warnings/errors

```
// implicit c++ conversions
void foo(double x) { ... }

int main() {
  bool b = true;
  int i = b;  // b promoted to int
  double d = 3.14 + i; // i promoted to double
  i = 2.718;  // 2.718 coerced/narrowed to int
  foo(i);     // i promoted to double
}
```

More Explicit	Less Explicit
Uglier, more verbose	Simpler, less verbose
Less likelihood of bugs	Higher likelihood of bugs
Less convenience	More convenience

Widening vs Narrowing

Widening and Narrowing Conversions

- **Widening** converts from a narrower type to wider type; result can fully represent the source type's values.
  - *value preserving* - converted value is always the same

- **Narrowing** converts wider to narrower type or between two unrelated types; the target type may lose precision or otherwise fail to represent the source type's values
  - The target type could be a subtype (like `long` and `short`), or the two types could also have a non-overlapping set of values (like `unsigned int` and `int`).

Widening and Narrowing Casts

- **widening cast (up-cast)** - casts subtype to supertype (eg: Student to Person)
  - never need explicit syntax for up-casts
  - always safe because supertype can do everything subtype can
- **narrowing cast (down-cast)** - casts supertype to subtype (eg: Person to Prof)
  - need explicit syntax
  - may fail if actual object isn't compatible with down-casted type

type promotion- implicit, widening conversion or cast

Perl supports type coercion: narrowing coercion from integer to boolean

```
$a = 5;
if ($a) {
    print("5 is true!");
}
```

Golang requires explicit conversions between even comparable types (like `int` and `float`).

```
func main() {
  var x int = 5
  var y float32 = 10.0
  var result float32

  result = float32(x) * y // doesn't work if we remove float32()
}
```

Checked and Unchecked

- in strongly-typed language, every conversion/cast with potential for issue is checked at runtime
- in weakly-typed language, some invalid conversions/casts may not be checked (leading to undefined behavior)

```
// Checked conversion (Java)
class Organism { ... }
class Alien extends Organism { ... }
class Dog extends Organism { ... }

public void play_time(Organism o) {
  Dog d = (Dog)o;    // can't cast; throws runtime exception
  d.play_fetch();
}
```

```
Alien a = new Alien();
play_time(a);
```

```
// Unchecked conversion (C++)
class Organism { ... }
class Alien: public Organism { ... }
class Dog: public Organism { ... }

void play_time(Organism* o) {
  Dog* d = (Dog *)o;   // no error
  d->play_fetch();     // undefined behavior
}

Alien *a = new Alien(...);
play_time(a);
```

Final Thoughts

- Type systems empower you to formalize a problem's structure into (user-defined) types.
- This allows the compiler to verify that structure, enabling you to write more robust software.